JavaScript with Classes

Diogo Sperb Schneider Eichert

JavaScript with Classes

10th edition

© 2012, 2023 Diogo Eichert

Preface

This book is dedicated to programmers who want to adopt the classical OOP paradigm in JavaScript, whether using or not external frameworks and other libraries, as this knowledge pertains to the language level. These techniques take some discipline to master at first, but they prompt the programmer to do more with less effort later on. They may also make the programmer feel more capable of tackling problems for which JavaScript would not even be considered as an option, in a first glance.

I've been working in projects which demand some massive amounts of JavaScript code. Coming from a Java and Objective-C reality, it was a tough task to accommodate my classical OOP needs, at first. After reading about existing frameworks and libraries which try to compensate for this problem (but don't directly address it) I had to resign to procedural code, until I learned my way to apply classical OOP techniques to this one-of-a-kind language. And then, after some hard work, it paid off.

The main inspiration for this material was the seminal OOP With ANSI-C book*, which shows how it's possible to write object-oriented code in plain C. It made me realise OOP was really more a paradigm than just preprocessing and language syntax.

There's a common misconception that adopting classes in a project should be an all-or-nothing decision, but it doesn't have to be like that. May the classes empower whatever you do, so use them well.

By the time of the original writing of this material, it was already possible to read a lot about the ECMAScript 6 specification, which should natively support classes to some extent. But even now that it's out, there won't be full support for it anytime soon, particularly in old interpreters out in the wild, which can take some time to be phased out. Also, one might not want to adopt the whole new set of changes promoted by ECMAScript 6 just to leverage the power of classes.

And even after the wide availability of ECMAScript 6, it is still possible to use the techniques displayed on this material to achieve better compatibility with legacy technology and also improve the encapsulation in natively supported ECMAScript 6 code.

Last but not least, I'd like to thank all who fostered the creation of this material, through advice, critic, influence, reviews or just motivational support. There would have been no JavaScript with Classes without them.

“Everything should be made as simple as possible, but not simpler”. - Albert Einstein

* Object-Oriented Programming With ANSI-C, by Axel Schreiner.

Table of Contents

JavaScript with Classes

- Preface
- Table of Contents
- 1. Fundamentals
- 1.a. Scope
- 1.b. Strict Mode
- 1.c. Closures
- 1.d. Arguments
- 1.e. Objects
- 1.f. Context
- 2. Classical Object Orientation
- 2.a. Classes
- 2.b. Constructors
- 2.b. Methods
- 2.c. Inheritance
- 2.d. Composition
- 2.e. Polymorphism
- 2.f. Private Members
- 3. Appendix
- 3.a. ModelViewController Pattern
- 3.b. Exporting and Importing Files
- 3.c. TDD (TestDriven Development)
- 3.d. ECMAScript 6
- 4. Conclusion
- 5. About the Author

1. Fundamentals

In this chapter, we'll see some very fundamental concepts upon which the more advanced techniques are built upon. It should be safe to skim the whole chapter if you already understand the most primitive building blocks of JavaScript or if you are willing to simply use the techniques displayed in this material without much concern about how they work.

1.a. Scope

In this first Section, we'll see Scope, the concept responsible for the fundamental part of encapsulation. Think of Scope in JavaScript as several boxes, one inside another, like a Russian Matryoshka doll, but made of reflexive glass. Each box can see outside through all boxes which contain it, but it cannot see inside the boxes it contains. This concept might sound trivial for C programmers, because it works just the same way.

Let's take a look at the following code snippet:

```
a = 1

function alpha() {
   b = a + 1
   beta()

   function beta() {
      c = b + 1
   }

   var d = c + 1
}

alpha()
// beta() // throws a "not defined" exception
console.log(a)
console.log(b)
console.log(c)
// console.log(d) // throws a "not defined" exception
```

Analysing the code above, we have simple attributions, function declarations, a variable declaration and finally some output printing. Please note that two lines are commented out on purpose. That's because those lines would produce errors. Why? Let's analyse each one of those:

Calling "beta()" from the global scope would throw a "not defined" exception because it was declared inside the scope of “alpha()". Just like trying to access "d" from the global scope would throw a "not defined" exception, because it was declared specifically with the reserved word "var".

Ok, but why was it just fine to access "b" and "c" from the global scope? Because they were not specifically declared like "d", so the interpreter assumed you were referencing some variable from the global scope and automatically assigned them to the global scope.

What might look as a feature might in fact contribute to the production of very poorly written code. There’s no way to predict how many references to objects we have in the global scope until we read the whole program.

So how can we prevent this, besides manually checking the whole program for undeclared variables? That’s what we’ll cover in the next section.

1.b. Strict Mode

Strict Mode is a feature introduced in ECMAScript 5 which turns JavaScript into a more robust language. Among other things, it will force the interpreter to throw errors if the code tries to indirectly create global variables. Let’s take a look at this code:

```
"use strict”; // this scope (global, in this case) is now in strict mode
// a = 1; // the interpreter will no longer tolerate this
var a = 1;

function alpha() {
   // b = a + 1;
   var b = a + 1;
   beta();

   function beta() {
      // c = b + 1;
      var c = b + 1;
   }
}

alpha();
console.log(a);
// console.log(b); // will raise a "not defined" exception
// console.log(c); // will raise a "not defined" exception
```

As you can see, in strict mode we can no longer declare variables without "var", so we also can't accidentally create variables in the global scope within another scope, i.e. the scope of a function. So we can rely on strict mode to prevent such mistakes and be warned by the interpreter as soon as they're found.

1.c. Closures

Since each function has its own scope, we can and should use them to prevent polluting the global scope. So how can we do that? We can use a closure. Closures are one of the features of functions, anonymous or not, which turn its containing objects (functions or variables) invisible outside of it. In this special case, we'll even use a self-executing closure, so that it does what it does as soon as it's parsed. Let's see what it looks like:

```
(function () {
    "use strict";
    var a = 1;

    function alpha() {
        var b = a + 1;
        beta();

        function beta() {
            var c = b + 1;
        }
    }

    alpha();
    console.log(a);
})();
// console.log(a); // throws a "not defined" exception
```

The outer anonymous function which embraces all is our closure. If it was just a regular anonymous function, our code wouldn't run, because it would never be called. But, since we enclose it with parentheses and then call it with another pair of parentheses, we put it to work instantly.

In the above case, our global scope remains intact. Note the commented line, it would throw an error because "a" never existed in the global scope, it it hidden inside our main closure. Even our strict mode statement is inside the closure, so even pieces of non-strict code in the global scope will be harmed by any possible interference from our code. We'll use closures to define the scope of our classes as well as the scope of each

separate file.

1.d. Arguments

Another important thing we must understand how it works in JavaScript are the function arguments, because they are the basis for our method signatures once we go with OOP. Let's take a look at the following code:

```
(function () {
   "use strict";

   function whatever(/*anything*/) {
      for (var i = 0; i < arguments.length; ++i) {
         var argument = arguments[i];
         console.log(argument);
      }
   }

   function another(mandatory, optional) {
      optional = optional || "default";
      console.log("You provided some " + mandatory + " but the rest was just plain " +
optional + ".");
   }

   function yetAnother(object) {
      if (object.one) {
         console.log("Got one,");
      }

      if (object.two) {
         console.log("two,");
      }

      if (object.three) {
         console.log("three options!");
      }
   }

   whatever(1024 * 1024);
   whatever("one", "two", "three");
   whatever(["Alpha", "Beta", "Gamma"]);
   another("arguments");

   yetAnother({
      "one" : 1,
      "two" : 2,
      "three" : 3
```

```
    });
})();
```

The function "whatever()" has no parameters specified, though it will accept any number of any kind of parameters through the builtin "arguments" array that every JavaScript function possess.

Another method is shown in the next function, "another()", which specifies two named parameters. Note that the parameter called "optional" has a default value assigned to it in the very first line code inside that function. Should "optional" be undefined, null or simply omitted, the "default" value will be attributed to it.

Yet another way of passing more complex variations of parameters to a function is through an object, like shown in the "yetAnother()" function. The function can check for any kind of properties inside that object and then behave accordingly.

1.e. Objects

Almost everything in JavaScript is an object. The biggest difference between Java objects and JavaScript objects is that JavaScript objects are soft. They need no class to be instantiated from and they can completely change their shape in runtime. Let's have a look at the following snippet:

```
(function () {
    "use strict";

    var object = new Object();
    object.property = "a string";
    var outside = {}; // this is just like new Object();
    outside.middle = {};
    outside.middle.inside = "another string";

    var other = {
        "number" : 1,
        "string" : "yet another string",
        "array" : new Array(),

        "object" : {
            "array" : [] // this is just like new Array();
        }
    };

    console.log("object"); console.log(object);
    console.log("object.property.property");
```

```
    console.log(object.property.property);
    console.log(“outside");
    console.log(outside);
    console.log("other");
    console.log(other);
    /*
    object.property.property = “what?";
    console.log("object.property.property");
    console.log(object.property.property);
    object.property = {};
    object.property.property = “what?";
    console.log(“object.property.property");
    console.log(object.property.property);
    */
})();
```

In the above example we can see three distinct ways of instantiating objects while adding members to them. As you can see, there’s no such thing as a class yet. There’s much more on JavaScript fundamentals but that’s not necessary for what we’re trying to accomplish here. So I won’t address these, there’s already too much good material on those subjects out there.

1.f. Context

One of the most misunderstood concepts in JavaScript is called “context”, which is basically what object the “this” special pointer refers to. It must not to be confused with scope. Context is crucial to the OOP techniques in this material, so a few tricks must be kept in mind when we need to pass functions as arguments, usually when integrating OOP code with other basic JavaScript constructs such as event handlers and callbacks.

Suppose you want to call a method from your instantiated object within a timeout, you might be tempted to do it like this:

```
setTimeout(myObject.myMethod, 123);
```

But note that “setTimeout()” takes a function pointer as argument, not a function call. When it actually calls the target function, its context has been replaced by something else (usually the “window” object) and your method will be broken, since all references to “this” inside the method’s code won’t be pointing to the expected object. So in order to preserve your method’s context, you should always use helper functions to be passed as arguments instead:

```
setTimeout(onTimeout, 123);

function onTimeout() {
    myObject.myMethod();
}
```

2. Classical Object Orientation

So far we have only covered the basics of JavaScript. While they're good enough for writing procedural code, we're going to use them as the foundation for our object oriented code.
In this chapter we will see how we can implement or emulate the building blocks of classical OOP.

2.a. Classes

Let's start creating a class. Since JavaScript doesn't officially support classes* these are sometimes called "pseudo classes", but we'll just refer to them as classes for simplicity. Take a look at the following snippet, which represents a full class, in big picture fashion:

```
var MyClass = (function () {
    var CONSTANT = "constant";

    // public
    function MyClass(parameter) {
        this.property = parameter || “default";
    }

    MyClass.prototype.method = function (parameter) {
        return privateMethod(parameter) + " " + this.property;
    };

    // public static MyClass.staticMethod = function () {
        return “static";
    };

    // private static
    function privateMethod(parameter) {
        return CONSTANT + " " + parameter;
    }

    return MyClass;
})();

console.log(MyClass.staticMethod());
var myInstance = new MyClass("constructor");
console.log(myInstance.method("parameter"));
```

Confusing? Don't be scared, we'll detail each separate element in the following chapters.

2.b. Constructors

Take a look at the following snippet:

```
var MyClass = (function () {
   function MyClass() {
      this.field = "whatever";
   }

   return MyClass;
})();

var myInstance = new MyClass();
console.log(myInstance.field);
```

As you can see, declaring a class can be achieved with functions. The closure around it is simply for aesthetics right now, but it will help us keep our class organised soon. The inner function is our class constructor. We know it's a class because it's name starts with a capital letter (for convenience) though the comment right above it can also help its identification. But what really turns this simple function into a pseudo class is how we use it.

Inside the constructor function, we use "this" to reference the object, so we can add members, like properties. In this case, we added a property called "field". We also instantiate that class* through the "new" command, just like Java. And once we have a reference to the object, we can access its members.

* Actually "class" is a reserved word in JavaScript and there are plans for it's use in future versions of the language.

2.b. Methods

OK, since we now know how to declare classes with their own members we may want to add methods to them so they encapsulate their own functionality. For that, we'll use JavaScript prototypes. Let's see it in the next code snippet:

```
var Thing = (function () {
    function Thing() {
        this.property = "something";
    }

    Thing.prototype.getProperty = function () {
        return this.property;
    };

    Thing.prototype.setProperty = function (property) {
        this.property = property;
    };

    Thing.prototype.doSomething = function () {
        console.log("Doing " + this.property + ".");
    };

    return Thing;
})();

var thing = new Thing(); thing.doSomething();
```

Notice the getter/setter methods as well as the "doSomething" method, which consumes the property. Pretty easy, huh? With this simple approach we have actual classes with methods, so can benefit from Encapsulation, one of the pillars of OOP.

2.c. Inheritance

When it comes to code reusability, two concepts quickly come to mind: inheritance and composition. With inheritance, we have a parent class which is extended by other classes which inherit from it. That means basically having all of the functionality from the parent class, plus whatever, if any, other functionality on the inheriting class. Let's see how we can implement inheritance in JavaScript in the following code sample:

```
(function () {
    "use strict";
```

```
function main() {
   var grandfather1 = new Grandfather();
   var grandfather2 = new Grandfather();
   var father1 = new Father();
   var father2 = new Father();
   var son1 = new Son();
   var son2 = new Son();
   var son3 = new Son();
   grandfather2.setName(“Michael");
   father2.setName("Michael II”);
   son2.setName("Michael III”);
   son3.setName("Billy");
   console.log(grandfather1.getFullName());
   console.log(grandfather2.getFullName());
   console.log(father1.getFullName());
   console.log(father2.getFullName());
   console.log(son1.getFullName());
   console.log(son2.getFullName());
   console.log(son3.getFullName());
}
var Grandfather = (function () {
   function Grandfather() {
      this.surname = "Doe";
      this.name = "John";
   }

   Grandfather.prototype.getFullName = function () {
      return this.surname + ", " + this.name;
   };

   Grandfather.prototype.setName = function (name) {
      this.name = name;
   };

   return Grandfather;
})();

var Father = (function () {
   function Father() {
      Grandfather.call(this); this.name = "John II";
   };

   Father.prototype = Object.create(Grandfather.prototype);
   return Father;
})();

var Son = (function () {
```

```
    function Son() {
      Father.call(this);
      this.name = "John III";
    };

    Son.prototype = Object.create(Father.prototype);
    return Son;
  })();

  main();
})();
```

As you can see, we created three classes, Grandfather, Father and Son. Son extends (inherits from) Father, which extends Grandfather. So we can implement "getFullName()" and “setName()" only once in Grandfather and use it on instances of Grandfather, Father and Son. Please also note that we explicitly called the super constructor from the inherited class in the inheriting class constructor. This is necessary to enforce proper object initialisation, so we better start keeping that in mind as soon as possible.

Also mind the use of a main function that not only makes our code look more organised, it also guarantees that all methods have been defined before we actually instantiate an object from our class. The main() function convention is necessary because our methods are created by means of attributions, not like ordinary functions, which are parsed before everything else. The trick is just like that, call main() right before closing your main closure and make sure you start the flow of your program from within the main() function. It could be declared anywhere, I just wanted to declare it before everything else to show that it really does what it’s supposed to do.

In the first edition of this material, I used to recommend the implementation of inheritance through the use of Child.prototype = new Parent() instead of Child.prototype = Object.create(Parent.prototype) because the latter was only supported in newer JavaScript runtimes, but since then, those old browsers became less relevant in the current scenario. This new approach helps preventing undesired behaviour when inheriting from classes which make external calls in its constructor.

To override any inherited (super) method, we just declare it in the inheriting class, like this:

```
var Son = (function () {
  function Son() {
    Father.call(this);
    this.name = "John III";
    this.nickname = “Johnny”;
```

```
    };

    Son.prototype = Object.create(Father.prototype);

    // override
    Grandfather.prototype.getFullName = function () {
        return this.surname + ", " + this.name + " aka " + this.nickname;
    };

    return Son;
})();
```

And if we ever need to call the inherited (super) method, e.g. within an overridden method, we can do it like this:

```
    // override
    Grandfather.prototype.getFullName = function () {
        return Father.prototype.getFullName.call(this) + " aka " + this.nickname;
    };
```

2.d. Composition

Like mentioned before, there's another way of reusing code, by means of composition. With composition, we try to encapsulate functionality in classes which contain other classes which implement their own specific functionality. This must not be confused with regular Aggregation. An aggregation allows elements to live independently, while composition enforces a relation in which one element does not exist without the other. Let's see it in the excerpt below:

```
(function () {
    "use strict";

    // functions function main() {
        var son = new Son();
        console.log(son.tell());
    }

    var Family = (function () {
        function Family() {
            this.surname = “Doe";
            this.legacy = "DNA";
        }

        return Family;
    })();
```

```
    var Son = (function () {
        function Son() {
            this.age = "young";
            this.family = new Family();
        }

        Son.prototype.tell = function () {
            var result = "I might be " + this.age + ", but I still carry my family's " +
this.family.legacy + ". I'm a " + this.family.surname + ".";
            return result;
        };

        return Son;
    })();

    main();
})();
```

2.e. Polymorphism

JavaScript does not perform type checking in your objects, so benefiting from polymorphism is almost trivial, though it's in our hands to make sure our objects actually implement every expected method. We can use the inheritance technique shown before to achieve this, but keep in mind it can significantly increase complexity and code verbosity. It's usually simpler to just use good naming convention while unit testing ensures object compatibility.

2.f. Private Members

Since our class is nicely wrapped inside its own closure, we can add as many private members as we want. First, let's review why our public members are public, in the first place:

```
var MyClass = (function () {
    function MyClass() { }
    MyClass.prototype.method = function () { };
    MyClass.staticMethod = function () { };
    return MyClass;
})();
```

The outer MyClass is a reference to the class in the outside scope, so the rest of our code can actually use it. Since we add our public methods (either instance or static) to it, they're public too. Don't forget to return the

reference to our class constructor (the inner MyClass) at the end of the closure, so it's actually assigned to the outer reference.

At this point, we can assume everything else will be private, because it will be visible only inside our class' scope. Let's see what it would look like with the addition of some private members:

```
var MyClass = (function () {
   var CONSTANT = Math.PI;
   var privateVariable = CONSTANT;
   function MyClass() { }
   MyClass.prototype.method = function () { };
   MyClass.staticMethod = function () { };

   // private static
   function privateStaticMethod = function () { };
   return MyClass;
})();
```

3. Appendix

So, now that we know how to do proper OOP with JavaScript, what can we do with it? In this appendix we'll see a few use cases for what we have just learned.

3.a. ModelViewController Pattern

The ModelViewController (MVC) Pattern is an elegant way to encapsulate your model (business logic, data structures, etc.), your views (user interface) and the controller (an entity which puts it all together). We can easily leverage this pattern to build a single-page web application. This is an example of how we can implement this pattern with object-oriented JavaScript:

```
(function () {
    "use strict";
    var navigationController;
    var mainViewController;
    var secondViewController;
    var thirdViewController;

    // class NavigationController
    var NavigationController = (function () {
        function NavigationController(rootViewController) {
            this.viewControllers = [rootViewController];
            this.currentViewController;
        }

        NavigationController.prototype.push = function (viewController) {
            this.viewControllers.push(viewController);
        };

        NavigationController.prototype.showViewController = function (viewController) {
            if (this.currentViewController == viewController) {
                return;
            }

            for (var i in this.viewControllers) {
                if (this.viewControllers[i] == viewController) {
                    this.hideAllViewControllers();
                    viewController.show();
                }
            }
        };
```

```
    NavigationController.prototype.hideAllViewControllers = function () {
        for (var i in this.viewControllers) {
            var viewController = this.viewControllers[i]; viewController.hide();
        }
    };

    return NavigationController;
})();

// class ViewController
var ViewController = (function () {
    function ViewController(view) {
        this.setView(view);
    }

    ViewController.prototype.getView = function () {
        return this.view;
    };

    ViewController.prototype.setView = function (view) {
        this.view = view || null;
    };

    ViewController.prototype.show = function () {
        this.view.style.display = "block";
    };

    ViewController.prototype.hide = function () {
        this.view.style.display = "none";
    };

    return ViewController;
})();

// class MainViewController extends ViewController
var MainViewController = (function () {
    function MainViewController() {
        this.setView(document.getElementById("mainView"));
        this.backButton = document.getElementById("mainViewBackButton");
        this.nextButton = document.getElementById("mainViewNextButton");
        this.backButton.disabled = true;
        addEvent(this.nextButton, "click", onNextButtonClicked);

        function onNextButtonClicked() {
            navigationController.showViewController(secondViewController);
        }
    }
```

```
    MainViewController.prototype = Object.create(ViewController.prototype);
    return MainViewController;
})();

// class SecondViewController extends ViewController
var SecondViewController = (function () {
    function SecondViewController() {
        this.setView(document.getElementById("secondView"));
        this.backButton = document.getElementById("secondViewBackButton");
        this.nextButton = document.getElementById("secondViewNextButton");
        addEvent(this.backButton, "click", onBackButtonClicked);
        addEvent(this.nextButton, "click", onNextButtonClicked);

        function onBackButtonClicked() {
            navigationController.showViewController(mainViewController);
        }

        function onNextButtonClicked() {
            navigationController.showViewController(thirdViewController);
        }
    }

    SecondViewController.prototype = Object.create(ViewController.prototype);
    return SecondViewController;
})();

// class ThirdViewController extends ViewController
var ThirdViewController = (function () {
    function ThirdViewController() {
        this.setView(document.getElementById("thirdView"));
        this.backButton = document.getElementById("thirdViewBackButton");
        this.nextButton = document.getElementById("thirdViewNextButton");
        addEvent(this.backButton, "click", onBackButtonClicked);
        this.nextButton.disabled = true;

        function onBackButtonClicked() {
            navigationController.showViewController(secondViewController);
        }
    }

    ThirdViewController.prototype = Object.create(ViewController.prototype);
    return ThirdViewController;
})();

function addEvent(target, type, listener) {
    if (target) {
        if (target.addEventListener) {
            target.addEventListener(type, listener, false);
```

```
        } else if (target.attachEvent) {
            target.attachEvent("on" + type, listener);
        }
    } else {
        console.log("Can't addEvent(): target object is null.");
    }
}

function main() {
    mainViewController = new MainViewController();
    navigationController = new NavigationController(mainViewController);
    secondViewController = new SecondViewController();
    navigationController.push(secondViewController);
    thirdViewController = new ThirdViewController();
    navigationController.push(thirdViewController);
    navigationController.showViewController(mainViewController);
}

main();
})();
```

As you can see, we have a main function which starts it all, a simple helper function called addEvent which deals with the complexities of attaching event listeners to objects in different web browser implementations, a NavigationController, a base ViewController and a few other classes which extend it. There's not much here besides the code for navigation between views, but the intent was to keep it simple enough to be easy to catch. This is the view:

Of course, for that code to even run, we need our views as well:

```
<!DOCTYPE html>

<html>
  <body>
    <div id="mainView" style="display: none;">
      <h1>Main View</h1>
      <input id="mainViewBackButton" type="button" value="Back"/>
      <input id="mainViewNextButton" type="button" value="Next"/>
      <hr />
      <p>Main view content...</p>
      &alpha;<br />
      <hr />
      <p>page #1</p>
    </div>

    <div id="secondView" style="display: none;">
```

```
        <h1>Second View</h1>
        <input id="secondViewBackButton" type="button" value="Back"/>
        <input id="secondViewNextButton" type="button" value="Next"/>
        <hr />
        <p>Second view content...</p>
        &beta;<br />
        <hr />
        <p>page #2</p>
    </div>

    <div id="thirdView" style="display: none;">
        <h1>Third View</h1>
        <input id="thirdViewBackButton" type="button" value="Back"/>
        <input id="thirdViewNextButton" type="button" value="Next"/>
        <hr />
        <p>Third view content...</p>
        &gamma;<br />
        <hr />
        <p>page #3</p>
    </div>

    <script src="main.js"></script>
  </body>
</html>
```

Where each view is represented by a properly identified div tag. Please note they're all hidden by default to prevent them from showing up before we want that to happen.

3.b. Exporting and Importing Files

Now that we are writing structured, OOP JavaScript code, we can write complex applications, which may grow much larger than a simple web form validation function. Separating our classes in different files is an interesting option to keep them organised. So let's suppose we want to put some of our classes in a separate library file. We can proceed to that by making our code look like this:

```
// lib.js
(function () {
  "use strict";

  // class Helper
  var Helper = (function () {
    function Helper() {
      this.property = "whatever";
```

```
        }

        Helper.prototype.doIt = function () {
            console.log(this.property);
        };

        return Helper;
    })();

    // exports
    window.lib = {
        "Helper" : Helper
    };
})();

// main.js
(function () {
    "use strict";

    // imports

var Helper = window.lib.Helper;

    // class Controller
    var Controller = (function () {
        function Controller() {
            var helper = new Helper();
            helper.doIt();
        }
    })();

    // functions
    function main() {
        var controller = new Controller();
    }

    main();
})();
```

It's easy to see how one relate to each other. The library exports itself to the global context, creating a simple namespace that here we called just "lib" (but could be something more elaborate such as "org.institution.product") and publishing it's single class there. We could have published as many classes as we wanted. We could also keep a few classes unpublished, in case they only make sense in the scope of the lib.js file.

The class is then imported in main.js and used just like our class had been declared in the same file. Last, but not least, we must include these files in our HTML file in a special, though simple way for this to work:

```
<!DOCTYPE html>

<html>
  <head>
    <script src="lib.js"></script>
  </head>

  <body>
    <script src="main.js"></script>
  </body>
</html>
```

Note that lib.js has been included before main.js. That's because main.js imports lib.js, so we want the library to be available when main.js runs. This approach is totally safe, regardless of which file the browser loads first, because the JavaScript interpreter will parse them in the right order anyways.

We could have put both <script> tags in the <body>, but if we put our libraries in the <head> we give them a head start (pun intended) while the browser is parsing the HTML body. It is not advised though to put code that can manipulate the DOM in the <head> though, because it may lead to unnecessary rework from the browser's part while rendering our page. So it is the perfect place for libraries which don't even run before some other code calls them. And that's why main.js (or any code which actually starts doing something) should always be included right before the end of the <body> tag.

3.c. TDD (TestDriven Development)

As soon as a software project gets bigger, it becomes more dangerous to change and optimise code. It's easy to break a feature without even noticing it from the code perspective.
TDD comes as a helpful tool to avoid such regressions. So how can we benefit from TDD in JavaScript? Let's say we have a class called Point which implements, well... a Point:

```
var Point = (function () {
  function Point(x, y) {
    this.setX(x || 0);
    this.setY(y || 0);
```

```
    }

    Point.prototype.getX = function () {
        return this.x;
    };

    Point.prototype.getY = function () {
        return this.y;
    };

    Point.prototype.moveX = function (width) {
        this.setX(this.getX() + width);
    };

    Point.prototype.moveY = function (height) {
        this.setY(this.getY() + height);
    };

    Point.prototype.setX = function (x) {
        this.x = x;
    };

    Point.prototype.setY = function (y) {
        this.y = y;
    };

    return Point;
})();
```

And you need to add or optimise functionality to it. How can you make sure it still behaves as expected after refactoring it? You could create specs for it, ensuring its behaviour:

```
var PointTest = (function () {
    function PointTest() {
        var point;

        // no args constructor
        point = new Point();
        assert(0, point.getX());
        assert(0, point.getY());

        // all args constructor
        point = new Point(1, 2);
        assert(1, point.getX());
        assert(2, point.getY());

        // partial args constructor
```

```
point = new Point(3);
assert(3, point.getX());

// getters & setters
point.setAccelerationX(4);
assert(4, point.getAccelerationX());
point.setAccelerationY(5);
assert(5, point.getAccelerationY());
point.setSpeedX(6);
assert(6, point.getSpeedX());
point.setSpeedY(7);
assert(7, point.getSpeedY());
point.setX(8);
assert(8, point.getX());
point.setY(9);
assert(9, point.getY());

// multiple objects
var point1 = new Point(10, 11);
assert(0, point1.getAccelerationX());
assert(0, point1.getAccelerationY());
assert(0, point1.getSpeedX());
assert(0, point1.getSpeedY());
assert(10, point1.getX());
assert(11, point1.getY());
var point2 = new Point(12, 13);
assert(0, point2.getAccelerationX());
assert(0, point2.getAccelerationY());
assert(0, point2.getSpeedX());
assert(0, point2.getSpeedY());
assert(12, point2.getX());
assert(13, point2.getY());

// speed to angle
point = new Point();
point.setSpeedToAngle(1, 0);
assert(1, Math.round(point.getSpeedX()));
assert(0, Math.round(point.getSpeedY()));
point.setSpeedToAngle(1, 90);
assert(0, Math.round(point.getSpeedX()));
assert(1, Math.round(point.getSpeedY()));
point.setSpeedToAngle(1, 180);
assert(1, Math.round(point.getSpeedX()));
assert(0, Math.round(point.getSpeedY()));
point.setSpeedToAngle(1, 270);
assert(0, Math.round(point.getSpeedX()));
assert(1, Math.round(point.getSpeedY()));
```

```
        // get angle
        point = new Point();
        point.setSpeedX(1);
        point.setSpeedY(1);
        assert(45, point.getAngle());
        point.setSpeedX(1);
        point.setSpeedY(1);
        assert(135, point.getAngle());
        point.setSpeedX(1);
        point.setSpeedY(1);
        assert(135, point.getAngle());
        point.setSpeedX(1);
        point.setSpeedY(1);
        assert(45, point.getAngle());

        // speed to point
        point1 = new Point();
        point2 = new Point(100, 50);
        point1.setSpeedToPoint(2, point2);
        point2.setSpeedToPoint(2, point1);
        assert(100 / 150 * 2, point1.getSpeedX());
        assert(50 / 150 * 2, point1.getSpeedY());
        assert(100 / 150 * 2, point2.getSpeedX());
        assert(50 / 150 * 2, point2.getSpeedY());
    }

    return PointTest;
})();
```

While assert() is just a simple function which compares values with their expectations. It's implementation could be as simple as:

```
function assert(expected, value) {
    if (expected !== value) {
        var e = new Error();
        var error = "Test failed for value " + value + ", expected " + expected + " " +
e.stack;
        console.log(error);
    }
}
```

3.d. ECMAScript 6

So far we've learned how to leverage the power of the classical object-oriented programming paradigm in JavaScript ES5. Now that ES6 is pretty much common place, we can just throw all that knowledge to the bin, right? Not really.

ES6 introduces a better way of expressing classes in JavaScript, but it does not cover all. So you can (or must) still make use of some techniques displayed on JSwC.

One important thing before we decide to move on and refactor all of your ES5 code into ES6: it will become incompatible with ES5 code. So if we decide to move on and embrace ES6, we must refactor the whole code.

So, let's see some of the techniques that are still valid, like class constants. ES6 has no support for class constants as we saw in JSwC. Let's have a look at the following code:

```
var Thing = (function () {
  var CONSTANT = "constant";

  function Thing() {
    var local = CONSTANT;
  }

  return Thing;
})();
```

We can refactor this code using the new ES6 class construct, but in order to keep the class constant, we also keep the attribution with closure technique, like this:

```
const Thing = (function () {
  const CONSTANT = "constant";

  class Thing {
    constructor() {
      let local = CONSTANT;
    }
  }

  return Thing;
})();
```

This way, we can already make use of the class construct, which makes a lot of sense specially if we want to extend some other class, declare methods or use the getter and setter notation but still keep the class constants contained within the class scope without having to migrate them to the global scope, which could result into name collision with other classes.

Another technique we might want to keep around is method overloading. Let's have a look at these two classes in ES5 fashion:

```
function Parent() {
  this.value = true;
}

Parent.prototype.do = function () {
  return this.value;
};

function Child() {
  Parent.call(this);
  // do something else
}; Child.prototype = Object.create(Parent.prototype);

Child.prototype.do = function () {
  Parent.prototype.do.call(this);
  // do something else
};
```

This code declares a bare class and another class that extends the first one. Now with ES6 we call replace the constructor overloading with the neat super keyword, unfortunately it doesn't work with method overloading, so we might want to keep this technique around a little longer so that we don't lose functionality as we refactor our code into this:

```
class Parent {
  constructor() {
    this.value = true;
  }

  do() {
    return this.value;
  }
}

class Child extends Parent {
  constructor() {
    super();
    // do something else
  }

  do() {
    Parent.prototype.do.call(this);
    // do something else
  }
```

```
}
```

As you can see, we can use super in the constructor and also extend the class with the new notation, which makes its declaration a lot cleaner, but for method overloading, we still need to use the call technique. That is necessary because super is unfortunately not supported in non-constructor methods.

One common question from people new to JavaScript with classes is how can we achieve constructor overloading in ES6 without resorting to hacking with conditional statements. And the answer to that is actually simpler than we might expect, as demonstrated in the following example:

```
class Thing {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  static fromHash(hash) {
    return new this(hash.a, hash.b);
  }

  static fromJson(string) {
    return this.fromHash(JSON.parse(string));
  }
}

let thing = new Thing(1, 2);

// or

thing = Thing.fromHash({ a: 1, b: 2 });

// or even

thing = Thing.fromJson('{ "a": 1, "b": 2 }');
```

As seen above, the class constructor takes care of the general use case, in this scenario simply receiving two parameters and storing them as instance members. On top of that, one can implement factories as static methods in the class for the more specific use cases. Also in the example above, we have one factory to generate an instance of the class taking its parameters from a hash and another one from a JSON string, which also overloads the fromHash factory.

And by using this as a reference to the class itself, the code is generic enough to be extendable and portable between classes. Also less redundant and less error prone, since it's not necessary to write the class name multiple times.

4. Conclusion

So we have seen why and how it is possible to write well-structured, classical object oriented JavaScript code using an approach which employs pseudo classes as well as a practical use for these techniques. I hope you have enjoyed reading this material as much as I enjoyed writing it and more important than that, I really hope you can benefit from these practices in your everyday JavaScript coding.

5. About the Author

Diogo Eichert is a professional software engineer, programming computers since the mid 80s. In his free time, he is also the author of core2d, the open-source JavaScript multimedia engine and library, as well as a few published game titles. None of these would have been possible without using classes.

Contact information:
- email: diogoeichert@icloud.com
- website: https://diogoeichert.github.io